Detekcija sudara

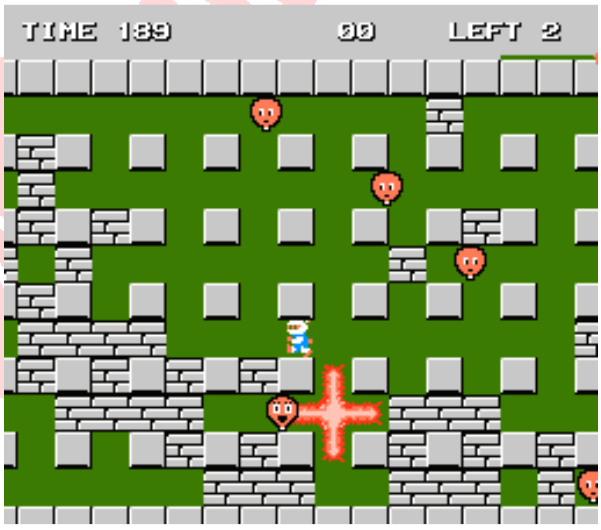
O detekciji sudara (collision detection) kao konceptu se može govoriti u dve ili tri dimenzije. Pošto je 2D animacija jedino moguća u Tkinteru – na nju ćemo se i fokusirati. Kako bismo osigurali da na određenom prostoru može biti samo jedan objekat, moramo implementirati detekciju sudara koja se bazira na geometriji tih objekata i njihovih pozicija. Postoji više tehnika kojima se mogu detektovati sudari; najviše su se razvijale zahvaljujući igračkoj industriji. Pošto ćemo detekciju sudara prikazati na primeru jednostavne igre gde je cilj sakupiti što više nasumično postavljenih poena (krugova) na kanvasu, o detekciji sudara ćemo pričati u kontekstu igara, ali se isti principi primenjuju i generalno na digitalnu 2D animaciju.

Tehnike rešavanja problema detekcije sudara

Pristup baziran na mreži (grid-based approach)

Prema ovom pristupu, svaka postavka (npr. trenutni nivo u 2D igri) na kanvasu bi bila podeljena na mrežu kvadrata, gde će svako polje ili sadržati objekat u igri (igrač, poeni, prepreke itd.) ili ne. Tačnije, svaki od tih objekata će zauzimati samo jedno polje.

Detekcija kolizije je u ovim slučajevima veoma jednostavna. Čitavu postavku zapravo možemo držati u dvodimenzionalnoj listi i pri svakom frejmu (Tkinter nam ne dozvoljava rad sa frejmovima (to nije isto što i tkinter.Frame objekat), pa možemo koristiti interni sat, tajmer ili pomeranje igrača i drugih elemenata) proveravati šta se u polju u koje treba preći nalazi.



Slika 9.1. Igra Bomberman (1983) – primer igre gde je detekcija sudara implementirana pomoću mreže[1]

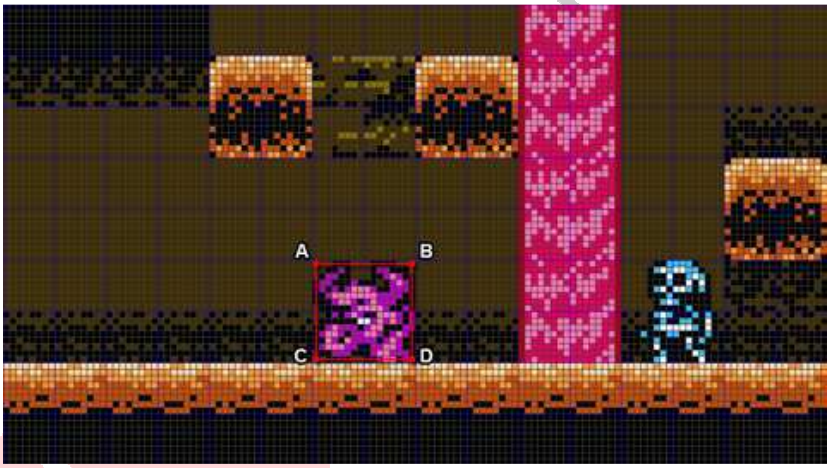
[1] https://en.wikipedia.org/wiki/Bomberman_(1983_video_game)

Pristup baziran na boji piksela

Ovakav pristup se koristio u začecima animacije i generalno igara na računarima, jer je nastao u periodu kada su mašine imale ograničen broj boja – ukupno 8 ili 16 boja. Detekcija je bila implementirana tako da je okruženje (površine za koliziju) bilo jedne boje, dok su igrač i drugi elementi igre bili drugačije boje, pa bi se pri svakom pomeranju elemenata igre proveravala boja piksela pored. Mana ovog pristupa bila je to što se boja piksela okruženja nije mogla koristiti nigde drugde.

Pristup baziran na pikselizovanju polja u mreži

Okruženje igre se deli na polja, a igrač može pristupiti samo predefinisanoj sekciji polja. Polja se mogu podeliti u grupe, recimo: pod/zemlja, nebo/pozadina, stepenice itd. Svako od tih polja se deli na piksele, gde se za svaki piksel definiše kojoj grupi pripada, pa se tako može desiti da je polje podeljeno dijagonalno, gde su ispod dijagonale stepenice, a iznad dijagonale nebo. Dalje, dodaju se i posebne tačke za proveru kolizije elementima u igri kako bi detekcija bila što tačnija, posebno na onim stranama tih elemenata za koje se pretpostavlja da će često biti u koliziji. Pa tako, ako se igrač nađe u takvom polju, dijagonalno podeljenom, u smeru gde je u sledećem koraku pozadina – treba spustiti igrača za jedan piksel.



Slika 9.2. Prikaz dodavanja posebnih tačaka za detaljniju proveru kolizije[2]

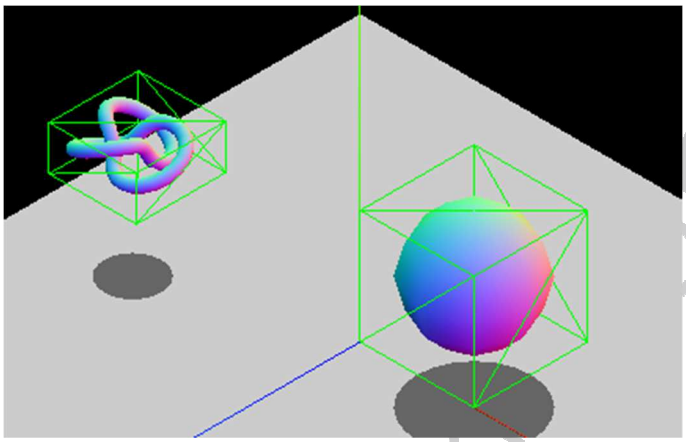
Maskiranje

Ova tehnika se zasniva na skladištenju dela mape ili čitave mape u memoriju u vidu crno-bele slike, gde je bela boja svaka površina nad kojom se može učiniti kolizija, a crna sve ostalo. Uz to, kopija objekta koji igrač može pomerati se takođe smešta u crno-beli format, gde su tačke za proveru kolizije bele, a sve ostalo crna površina. Pri pomeranju se konstantno na nivou piksela proverava da li dolazi spajanja dve površine – ako dolazi, onda je reč o koliziji. Na ovaj način se postiže to da se igrač može kretati slobodno i da tačnost bude na nivou piksela, ali zato ovakav pristup iziskuje veću iskorišćenost resursa.

[2] https://medium.com/@megacatstudios/smooth-jumping-for-platformers-3a5a7645ffc3

Granični okvir (bounding box)

Granični okvir elementa u 2D ili 3D sistemu je princip koji se primenjuje i u igrama i u animacijama. Reč je o iscrtavanju najmanjeg mogućeg pravougaonika kojim se može oivičiti predmet. U kontekstu kolizija, pri svakom frejmu se vrše proračuni da li se bilo koja dva granična okvira seku. Mana ovakvog pristupa je što mu efikasnost opada sa porastom kompleksnosti objekta.



Slika 9.3. Izgled graničnih okvira 3D objekata (objekti sa zelenim ivicama)[3]

Diskretna detekcija kolizije

Ova tehnika je dobila na značaju s pojavljivanjem prvih ozbiljnijih 3D igara i animacija, jer se pomoću nje može pratiti više objekata odjednom, kao i njihove trajektorije. Koncept je takav da se u svakom trenutku računaju granični okviri objekata u pokretu u trenutnom, kao i u narednom frejmu – AABB (axis-aligned bounding box) i svi granični okviri koji imaju zajedničku tačku se grupišu. Dalje, sa svakim od objekata u ovoj grupi vrši se kalkulacija kada bi potencijalno moglo doći do kolizije i koja bi mogla biti njihova trajektorija nakon kolizije. Iz ovoga vidimo da je nakon te kalkulacije ponovo potrebno izvršiti kalkulaciju i odrediti gde će se objekat dalje kretati i da li će opet doći do sudara. Koliko će se iteracija ovih kalkulacija izvršiti je na programeru. Ovakav pristup se retko koristi u 2D animaciji i igrama.

Spekulativni pristup

Prethodna metoda ne uspeva da reši problem kolizije veoma brzih objekata, koji se kreću mnogo brže nego što ima frejmova, pa se zbog usporene kalkulacije i toga što se kalkuliše samo početna i finalna pozicija – dešava da objekat prođe kroz površinu. Na primer, ako je reč o objektu veličine jednog piksela koji se kreće 100 piksela po frejmu, koji ide ka objektu debljine pet piksela – on će jednostavno proći kroz njega jer nema dovoljno vremena za preciznu kalkulaciju. Ovo se rešava tako što se pored startne i finalne pozicije računaju i sve pozicije između.

[3] https://developer.mozilla.org/en-US/docs/Games/Techniques/3D_collision_detection

Pitanje

Tehnika iscrtavanja najmanjeg mogućeg pravougaonika kojim se može oivičiti predmet se zove:

- bounding grid
- **bounding box**
- bounding cube

Objašnjenje:

Tačan odgovor je da se tehnika iscrtavanja najmanjeg mogućeg pravougaonika kojim se može oivičiti predmet zove bounding box. Mana ovakvog pristupa jeste to što mu efikasnost opada sa porastom kompleksnosti objekta.

Tkinterov pristup i kreiranje igre

Pošto Tkinter nije zamišljen sa animacijom na umu – nemamo mnogo izbora za rešavanje problema detekcije sudara. Jedan od pristupa koje možemo koristiti jeste da se prilikom svake komande korisnika (pomeranje levo, desno, gore, dole) proverava da li postoji tačka na kanvasu u kojoj postoji još jedan objekat pored našeg. Takođe, pored ovog rešenja, umesto trenutnih koordinata našeg objekta, koristićemo ugrađenu metodu `bbox()`, koja nam daje koordinate graničnog okvira. Naime, ona vraća listu koordinata objekta za po jedan piksel više po x i y osi nego što je `coords()` metoda vratila. Sada ove koordinate možemo proslediti metodi `find_overlapping()`, koja vraća n-torku svih objekata (njihovih id-jeva na kanvasu) koji se nalaze u objektu opisanom tim koordinatama. Pa tako, ako nam ta metoda vrati samo jedan id, to znači da nema kolizije. Ako nam vrati n-torku sa više elemenata, znači da ima kolizije. U toj n-torci će u našem slučaju nulti element uvek biti element na kanvasu koji igrač kontroliše. Implementacija detekcije kolizije se nalazi u `overlapping_helper()` funkciji:

Implementacija funkcije kojom se realizuje detekcija sudara

```
def overlapping_helper():
    s = canvas.bbox(player)
    overlapping_result = canvas.find_overlapping(s[0], s[1], s[2], s[3])


    if len(overlapping_result) > 1:
        canvas.delete(overlapping_result[1])
        global score
        score += 1
        score_string_var.set("Score: {}".format(score))
```

Takođe, svaki id koji nije nulti element je zapravo element sa kojim je nastala kolizija i njega ćemo ukloniti sa kanvasa i povećati rezultat igrača za jedan. Rezultat igrača će se pratiti u Tkinter `StringVar()` promenljivoj `score_string_var`, koja će biti povezana sa labelom ispod kanvasa.

Implementacija setup() funkcije kojom postavljamo poene po kanvasu

```
def setup():
    for i in range(opponents):
        random_color = get_random_color()
        x = randint(10, WIDTH - 15)
        distance = randint(5,30)
        y = randint(10, HEIGHT - 15)
        canvas.create_oval(x, # x0
        y, # y0
        x+distance, # x1
        y+distance , # y1
        fill = random_color)
```

Poeni koje igrač sakuplja su manji krugovi koji se generišu nasumično funkcijom `setup()`. U njoj se koristi globalna konstanta `opponents`, koja označava koliko će poena biti na mapi. Za svaki od tih poena, kreira se ovalni objekat na kanvasu, sa nasumičnim koordinatama, veličinom i bojom. Nasumična boja se generiše kao string u heksadecimalnom formatu funkcijom `get_random_color()`. Za generisanje nasumičnog celog broja koristimo `randint()` funkciju iz `random` biblioteke, kojoj uvek moramo proslediti dva broja: početne i krajnje vrednosti opsega iz kog želimo da izaberemo jedan nasumičan ceo broj. Takođe, ova funkcija vraća tip int, odnosno nasumično izabran ceo broj. Preko ove funkcije uzimamo tri broja u opsegu od 0 do 255, koji na numerički način predstavljaju crvenu, zelenu i plavu (RGB sistem boja). Nakon što dobijemo te numeričke vrednosti, prebacujemo ih u heksadecimalni format koristeći `X` kao `format specifier` i `02`, što označava da heksadecimalni broj želimo da prikažemo kao dvocifren.

Implementacija funkcije kojom generišemo nasumičnu boju poena

```
def get_random_color():
    r = lambda: randint(0,255)
    return '#%02X%02X%02X' % (r(),r(),r())
```

Poslednje što možemo dodati ovoj igri je tajmer. Tajmerom je igrač ograničen na predefinisan broj sekundi da skupi sve poene na mapi. Implementiraćemo ga u `update_clock()` funkciji. Merenje proteklog vremena činimo tako što uzimamo vreme sa početka aplikacije – `time.time()` metodom. To vreme uzimamo kao referentno i u svakom narednom pozivu `update_clock()` funkcije ga oduzimamo od trenutnog kako bismo dobili razliku, koja je takođe izražena u sekundama (linija: `seconds_diff = int(now) - int(start_time)`). Kada ta razlika dostigne predefinisanu vrednost koja predstavlja ukupno trajanje runde, igra se završava. U `update_clock()` funkciji smo takođe koristili `.configure()` metodu nad kanvas objektima, kojom nakon inicijalizacije možemo menjati atribute. Ako smo prilikom inicijalizacije postavili tekst labele `label_test` na `Hello`, u toku programa taj tekst (a i bilo koji drugi atribut) možemo promeniti pozivom `.configure()` metode: `label_test.configure(text= 'Hello World')`.

Na kraju, rezultat je prikazan korisniku u vidu `messagebox.showinfo` dijaloga.

Čitav kod igre se nalazi ispod:

Kompletan kod igre

```
import tkinter as tk
from tkinter import messagebox
from random import randint
import time
root = tk.Tk()

WIDTH = 500
HEIGHT = 500
label = tk.Label(root, text="Use arrow keys to move", font = 'Bold')
label.pack()
canvas = tk.Canvas(root, width=WIDTH, height=HEIGHT)

canvas.pack()


x1 = WIDTH / 2
y1 = HEIGHT / 2



player = canvas.create_oval(x1, y1, x1 + 40, y1 + 40, fill = 'green', width
= '5', outline = 'Blue')

time_label = tk.Label(root,text = 'Time left (seconds): ', fg = 'Green',
font = 'Bold')
time_label.pack()

score = 0
start_time = 0
duration = 10
opponents = randint(8,20)

score_string_var = tk.StringVar()
score_string_var.set('Score: {}'.format(score))


score_label = tk.Label(textvariable = score_string_var, font = 'Bold')
score_label.pack()

timer = None
def update_clock():
    now = time.time()
    global timer
    timer = root.after(1000, update_clock)
    seconds_diff = int(now) - int(start_time)
    time_label.configure(text = "Time left(seconds): {}".format(duration-
seconds_diff))
    if seconds_diff > duration:
        for canvas_widget in canvas.find_all():
            if canvas_widget != player:
                canvas.delete(canvas_widget)
        root.after_cancel(timer)
        time_label.configure(text = 'Game over, time up!', fg = 'red', font
= 'Bold')
        global score
```

```
        messagebox.showinfo(title=None, message="Well done, your score is:
{} out of {}!".format(score, opponents))




def overlapping_helper():
    s = canvas.bbox(player)
    overlapping_result = canvas.find_overlapping(s[0], s[1], s[2], s[3])


    if len(overlapping_result) > 1:
        canvas.delete(overlapping_result[1])
        global score
        score += 1
        score_string_var.set("Score: {}".format(score))


def move(event):
    overlapping_helper()

    if event.keysym == "Left":
        current_coords = canvas.coords(player)
        if current_coords[0] <= 0:
            canvas.move(player, WIDTH, 0)
        else:
            canvas.move(player, -15, 0)
    elif event.keysym == "Right":
        current_coords = canvas.coords(player)
        if current_coords[0] >= WIDTH:
            canvas.move(player, -WIDTH, 0)
        else:
            canvas.move(player, 15, 0)
    elif event.keysym == "Up":
        current_coords = canvas.coords(player)
        if current_coords[1] <= 0:
            canvas.move(player, 0, HEIGHT)
        else:
            canvas.move(player, 0, -15)

    elif event.keysym == "Down":
        current_coords = canvas.coords(player)
        if current_coords[1] >= HEIGHT:
            canvas.move(player, 0, -HEIGHT)
        else:
            canvas.move(player, 0, 15)

def setup():
    for i in range(opponents):
        random_color = get_random_color()
        x = randint(10, WIDTH - 15)
        distance = randint(5,30)
        y = randint(10, HEIGHT - 15)
        canvas.create_oval(x, # x0
        y, # y0
        x+distance, # x1
        y+distance , # y1
```

```
        fill = random_color)

def get_random_color():
    r = lambda: randint(0,255)
    return '#%02X%02X%02X' % (r(),r(),r())

setup()
start_time = time.time()
update_clock()
root.bind("<KeyPress>", move)
root.mainloop()
```

Ako se polaznik opredeli za produbljivanje ove teme, preporuka je korišćenje PyGame biblioteke, koja je kreirana upravo sa idejom razvoja 2D igara, pa tako taj proces znatno olakšava.

Izmeniti igricu tako da igrač ima oblik kvadrata kao i oblici koje je potrebno skupiti.

Rezime

- Detekcija sudara u pristupu baziranom na mreži (grid-based approach) vrši se tako što se trenutni nivo u igri podeli na polja (matricu / 2D listu). U tom slučaju se u polju može naći samo jedan element.
- Detekcija kolizije u pristupu baziranom na broju piksela bila je implementirana tako da je okruženje (površine za koliziju) bilo jedne boje, dok su igrači i drugi elementi bili drugih boja; pri svakom pomeranju elemenata igre, proveravala bi se boja piskela pored.
- U pristupu baziranom na pikselizovanju polja u mreži, polja su se unutra delila na piksele, pa se time omogućilo i parcijalno korišćenje polja, tj. da se u istom polju može naći više elemenata. Svim objektima određenim za koliziju su se dodavale posebne tačke za koliziju, koje su se u svakom trenutku proveravale.
- Maskiranje se zasniva na skladištenju dela ili čitave mape u vidu crno-bele slike u memoriju, gde je bela boja svaka površina nad kojom se može učiniti kolizija, a crna sve ostalo. Pri pomeranju se konstantno na nivou piksela proverava da li dolazi do spajanja dve površine – ako dolazi, reč je o koliziji.
- Granični okvir elementa u 2D ili 3D sistemu je princip koji se primenjuje i u igrama i u animacijama. Reč je o iscrtavanju najmanjeg mogućeg pravougaonika kojim se može oivičiti predmet.
- Diskretna detekcija kolizije se bazira na kalkulaciji trenutne pozicije elementa u pokretu i njegove krajnje pozicije. Nakon što se izvrši kolizija, ova kalkulacija se ponavlja i opet u računicu uključuju brzina i pravac kretanja, kao i okruženje.
- Spekulativni pristup detekciji kolizije unapređuje prethodnu tehniku tako što pored startne i krajnje pozicije računa i sve pozicije između na trajektoriji.
- U Tkinteru, metoda bbox() nam daje koordinate graničnog okvira vidžeta i vraća listu koordinata objekta za po jedan piksel više po x i y osi nego što je coords() metoda vratila.
- Metoda find_overlapping() vraća n-torku svih objekata (njihovih id-jeva na kanvasu) koji se nalaze u objektu opisanom prosleđenim koordinatama.